Licence Génie Mathématique et Informatique
Université Paris IX Dauphine
Rapport de Stage
Mai – Août 2002

Code : 2/01/F/ENT

Une Messagerie dans le Shell

Carl Seleborg

Enetshare
Maître de stage : M. Denis Gueyffier

Conception et Développement — APIs Shell & Win32 — COM/ATL — Ergonomie du logiciel

Une Messagerie dans le Shell

Rapport de stage
6 Mai - 30 Août 2002
38 pages

Enetshare
64, bd Arago
75013 Paris
France

Responsable de stage : Denis Gueyffier
Tel : 01 49 23 97 54 Mobile : 06 61 76 34 16
E-mail : denis@alum.mit.edu

Certaines informations concernant le projet de logiciel de messagerie de la société Enetshare sont encore, à l'heure de la rédaction de ce rapport, soumises à un accord de confidentialité entre le stagiaire et l'entreprise, principalement pour des raisons de propriété industrielle. Par conséquent, certains aspects du logiciels seront volontairement — et malheureusement — peu détaillés et/ou peu évoqués.

Table des matières

Table des figures

Préambule

Ce document présente le stage effectué par l'auteur au sein de l'entreprise Enetshare durant les mois de mai à août 2002, au terme de l'année de licence Génie Mathématique et Informatique, à l'Université Paris IX Dauphine.

Enetshare est une entreprise naissante : elle avait un mois lors du début du stage. Elle a été créée par Denis Gueyffier pour concrétiser une idée qu'il a eu lorsqu'il travaillait au MIT, aux Etats-Unis. M. Gueyffier souhaitait proposer une nouvelle approche de la communication sur Internet, tant sur le plan des techniques réseau utilisées que sur l'ergonomie du logiciel. Il souhaitait gommer la frontière entre messagerie instantanée (type MSN Messenger ou ICQ) et le courrier électronique, et surtout changer la manière visuelle de représenter un correspondant. Un tel correspondant, ou contact, ne doit plus être une adresse e-mail ou un surnom de messagerie, inscrit dans une liste, à l'intérieur de la fenêtre du logiciel utilisé, mais doit plutôt être représenté visuellement par une icône distincte. Le correspondant, pour Enetshare, sera sur le bureau ou ne sera pas.

La plus grande partie du stage s'est dès lors résumée à trouver à la réponse à la question suivante : comment créer un logiciel de messagerie qui ne montre à l'utilisateur que ce qu'il a l'habitude de voir, c'est-à-dire le bureau Windows ?

Durant quatre mois, le travail a été une profonde plongée au cœur de la programmtion Windows. Les principaux outils de travail furent Visual C++, l'environnement de développement phare de Microsoft, Windows 2000 et la gigantesque bibliothèque MSDN qui contient la documentation technique de toutes les technologies Microsoft. Les découvertes furent nombreuses, et la plus importante fut sans doute le modèle de composant COM†, un modèle objet inventé par Microsoft, et qu'on nomme aussi OLE ou ActiveX (à peu de choses près). D'une façon plus générale, ce stage a également été une grande initiation à la programmation Windows, sans toutefois aborder les MFC, la grande bibliothèque objet de Microsoft, destinée a remplacer la programmation C, souvent laborieuse, pour les applications Windows.

Le temps du stage n'étant pas suffisant pour développer une application intégrée à ce point à Windows, mon rôle fut de "préparer le terrain" : je devais explorer les solutions possibles et éventuellement, en trouver les limites. Je devais aussi proposer une conception du logiciel qui soit suffisamment pérenne pour que d'autres développeurs puissent continuer sur cette base après moi. J'ai laissé les bases d'une applications, une petite "boite à outils" sous forme de DLL† contenant des dizaines de classes utilisables pour le projet, et surtout un début d'implémentation de serveur COM, qui contiendra, à terme, tous les objets nécessaires à l'intégration dans Windows. Tout ce code est assez largement documenté.

Les termes marqués d'un symbole, comme ceci†, sont repris dans le glos-

saire à la fin de ce document. Il s'agit pour la plupart de termes techniques. L'annexe en fin de document montre un exemple de définition de classe (la classe `CObjectID` en C++, avec la documentation Doxygen incorporée dans le code.

1 L'Entreprise

1.1 De l'idée à l'entreprise

C'est alors qu'il était encore enseignant-chercheur au MIT, aux Etats-Unis, que Denis Gueyffier a eu l'idée de son logiciel de messagerie. C'était une idée innovante, qui proposait une nouvelle manière de présenter la messagerie électronique à l'utilisateur. Toute l'idée reposait sur un concept d'interface utilisateur intuitive : le logiciel devait être très simple à utiliser. Selon les mots de M. Gueyffier : "*... tellement simple que même ma grand-mère pourra l'utiliser*". C'est vous dire !

Quelques temps après son retour en France, après avoir travaillé comme directeur technique pour une entreprise française, M. Gueyffier décide de tenter sa chance en concrétisant son projet. Ce qui devait nécessairement passer par la création d'une entreprise pour l'héberger. Après avoir hésité quelques temps sur le nom (au début, elle devait s'appeller *The Drag-n-Drop company*, c'est finalement le nom d'Enetshare qui fut choisi.

L'entreprise est "née" officiellement en avril 2002. Il s'agit d'une EURL, c'est-à-dire d'une SARL avec une seule personne à sa tête, au capital de départ de 8000 euros. Etant donné qu'elle n'a pas encore de produit, il n'y a pas encore d'entrées d'argent — ce qui implique qu'il est nécessaire que le logiciel Enetshare passe bientôt devant les investisseurs, car les réseves d'argent ne sont, hélas ! pas inépuisables.

C'est avec l'aide de quelques intervenants extérieurs, travaillant pour Enetshare au titre de consultants (cf paragraphe 3.1, p. 28), car M. Gueyffier n'a pas les moyens pour le moment pas embaucher quelqu'un à temps plein, que le projet va démarrer et que seront fixées les grandes orientations. Et quelques temps après, j'arrivais dans l'entreprise comme stagiaire-développeur pour produire un prototype du logiciel, destiné à être montré aux investisseurs.

1.2 A la recherche d'investissements

Aux Etats-Unis, la moindre idée quelque peu originale peut assez facilement générer des investissements. C'est un pays où les hommes d'affaires ont le sens du risque, de l'innovation et de l'entreprise. A tel point même qu'il faut toujours être extrêmenent prudent lorsqu'on parle d'une idée, et il faut en général se garder de l'exposer, sous peine de se la faire voler : les américains considèrent que dès l'instant où une idée est dite à voix haute, elle appartient au domaine publique.

En France, les choses sont bien différentes, et les investisseurs sont autrement plus frileux. M. Gueyffier a eu, et a toujours, toutes les peines du monde à trouver des investissements pour son projet. Plus personne, après le fiasco de la Net-Economie, ne veut investir dans un projet informatique dont on n'est pas sûr à 100% qu'il réussira.

C'est en gagnant le concours national d'aide à la création d'entreprises de technologies innovantes de l'ANVAR, l'agence française de l'innovation, que Denis Gueyffier a pu recevoir suffisamment d'argent pour lancer son projet et pour financer les premiers pas de l'entreprise (matériel informatique, consultants, connexion Internet, etc...). C'est d'ailleurs par la voie de subventions, plutôt que par les investissements, que M. Gueyffier obtiendra peu à peu les fonds nécessaires à la survie du nouveau-né. Mais ce n'est pas suffisant, et bientôt il faudra trouver le soutien d'investisseurs réels pour pouvoir faire éditer le logiciel, et pour en faire la promotion dans les médias, sans quoi le lancement pourrait bien être un échec.

1.3 Perspectives d'avenir

Pour Denis Gueyffier, Enetshare est une sorte de défi : "*Je voulais montrer au monde qu'il était possible de mettre des contacts sur le bureau*". Mais ce n'est pas vraiment une fin en soi. Si le projet échoue, il échouera, c'est tout. C'est d'ailleurs ce qui risque d'arriver si les investissements se font attendre. Si jamais le projet échoue, M. Gueyffier souhaite faire don du code source à la communauté Open Source, afin que le logiciel puisse tout de même se développer, ce qui est envisageable, car il se base sur un certain nombre de standards, la plupart desquels sont libres, comme Jabber ou SSL.

Mais si l'argent rentre, et qu'il est possible à Enetshare de continuer le développement de son produit, Denis Gueyffier souhaiterait trouver un *incubateur*, une société ou un organisme qui héberge de petites entreprises en proposant un ensemble de services (locaux, connexions Internet, salles de conférences, etc...). A l'heure où ce rapport fut rédigé, M. Gueyffier était en pourparlers avec l'INRIA Transfer, l'incubateur de l'INRIA, qui s'intéresse plus particulièrement aux projets technologiques innovants. En terme de personnel, il faudra bientôt embaucher un ou plusieurs développeurs, capables de finaliser le produit, et sans doute aussi un cadre commercial qui saura le vendre aux entreprises, cible actuelle d'Enetshare.

2 Le travail effectué

Cette section présente le travail effectué d'un point de vue technique.

2.1 Présentation du logiciel

Enetshare (c'est le nom de logiciel pendant la phase de développement ; ce nom pourrait changer par la suite) est un logiciel de messagerie, pour l'instant prévu pour Windows[1]. Destiné à être simple d'usage, il tent à laisser l'utilisateur travailler dans un environnement qu'il connaît, le Shell, plutôt que d'imposer ses propres fenêtres pour tout. En effet, comme tout logiciel de messagerie qui se respecte, Enetshare est doté d'un carnet d'adresses, appelé *Roster*†, où l'utilisateur peut cataloguer ses contacts. Les "ensembles" de contacts sont appelés des *familles*, et une famille peut contenir plusieurs types d'*objets* différents (une famille peut donc contenir des sous-familles et des contacts). Il peut assigner une icône prédéfinie à certains contacts, faisant de ceux-ci des contacts "privilégiés" ou "favoris", qu'il pourra placer sur son bureau pour les joindre encore plus facilement. Ce carnet d'adresses s'inscrit, au même titre que les imprimantes ou les ordinateurs du réseau, dans l'arobrescence du Shell de Windows, ce qui signifie que l'utilisateur manipule son carnet d'adresses dans la même application que ses fichiers, à savoir Explorer.

Enetshare est un logiciel de messagerie hybride, qui repose sur un serveur Jabber, un serveur de messagerie instantanée Open Source. Le principe est d'envoyer le message par la voie la plus rapide possible : si la personne à laquelle l'utilisateur veut envoyer un message est un utilisateur Enetshare et qu'il est connecté au moment de l'envoi, le message partira comme dans le cas d'une messagerie instantanée, et les deux protagonistes pourront converser en direct en voyant les messages dès que ceux-ci sont envoyés. Si, en revanche, l'autre utilisateur n'est pas connecté, ou s'il ne s'agit pas d'un utilisateur Enetshare, le message partira comme un message e-mail ordinaire.

Lorsque l'utilisateur reçoit un message (qu'il s'agisse d'un message instantané ou d'un e-mail), une icône située dans la barre des tâches, à côté de l'horloge, le prévient de l'évènement. Il peut alors répondre directement en cliquant sur cette icône. Une petite fenêtre d'édition apparaît dans laquelle l'utilisateur peut taper son texte — s'il s'agit d'une conversation en direct, il y verra également les réponses de son interlocuteur. Il peut également envoyer à son correspondant des fichiers ou des images, par une simple opération de Drag-n-Drop sur cette fenêtre.

Lorsque le destinataire du message est également un utilisateur Enetshare, tout message envoyé est crypté : ceci assure une bonne sécurité des transmissions, et permettra à la société Enetshare, sur le plan stratégique,

[1] Bien que ce logiciel soit fortement intégré à Windows quant à son interface graphique, une version pour Linux/Unix est en projet.

de convaincre de potentiels clients professionnels, qui pourront enfin proposer à leurs employés un système de messagerie instantanée dont les messages ne transiteront pas par des serveurs étrangers, comme c'est le cas pour ICQ ou pour MSN Messenger, le service de messagerie de Microsoft.

2.2 L'intégration du logiciel à Windows

A l'origine, M. Gueyffier souhaitait avoir « des icônes sur le bureau ». C'était le principe de base de l'interface du logiciel. Après avoir exploré divers moyens d'obtenir ce résultat (en particulier au moyen des *ActiveZone*, cf. paragraphe 2.4.1, p. 24), nous avons fini par être convaincus que le seul réel moyen de procéder correctement était d'intégrer le logiciel au Shell de Windows.

2.2.1 Présentation du Shell

Le Shell de Windows est l'ensemble des éléments visuels de l'interface de Windows (à partir de la version Windows 95) que l'utilisateur voit comme *intrinsèque* au système. Il s'agit, entre autres, du Bureau, du Poste de Travail, de Windows Explorer, du paneau de configuration, etc... mais aussi des menus déroulants qui apparaissent lorsqu'on clique avec le bouton droit de la souris sur une icône, des mécanismes de *Drag-n-Drop*†, et d'autres encore. Le Shell propose également une organisation virtuelle de ses divers objets (le Shell contient beaucoup d'objets !). Cette organisation est l'arborescence qu'on voit dans la partie gauche de l'Explorateur de fichiers : on l'appelle le *Namespace* du Shell. Cf. figure 1.

La racine du Namespace du Shell, ou *Shell Namespace* en anglais, est le bureau. D'une manière générale, tous les nœuds de cette arborescence qui contiennent d'autres nœuds sont appelés *Folders*. Les feuilles de l'arborescence sont appelées *Items*. En particulier, dans le cadre du Shell Namespace, les répertoires du système de fichiers sont des Folders, et les fichiers sont des Items. Mais ce n'est qu'un cas particulier : parmi les Folders du Shell, on compte aussi le Poste de Travail, le Panneau de Configuration, le dossier Imprimantes, le Planificateur de Tâches, etc.... Quelques exemples d'Items sont les imprimantes, les applets du panneau de configuration ou encore les tâches planifiées. Seuls les Folders apparaissent dans la partie gauche de Windows Explorer.

L'utilisateur averti aura vite noté que cette organisation ne correspond pas à l'organisation des fichiers du disque. En effet, sur le disque il n'y a pas de répertoire contenant des imprimantes. Il n'y a pas non plus de répertoire contenant des fichiers qui contiendraient les ordinateurs du réseau. Cette arborescence est donc virtuelle. Il ne s'agit en réalité que d'une structure mise en place par le Shell et qu'il est possible d'étendre avec ce qu'on appelle les *Namespace Extensions*, qui rajoutent des Folders dans le Shell. Ainsi, par

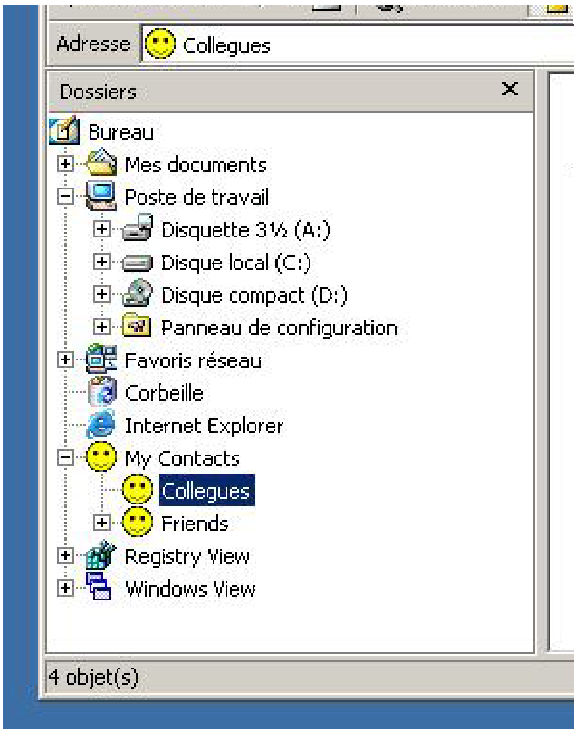

Fig. 1 – Le Namespace du Shell

exemple, le Poste de Travail n'est qu'un Namespace Extension qui insère dans l'arborescence du Shell l'ensemble des fichiers et répertoires des disques de l'ordinateur.

Les Namespace Extensions sont donc des sortes de *plug-ins* qu'Explorer (explorer.exe est le nom du programme qui crée le Shell, et par extension, le Shell s'appelle Explorer) charge lors de son démarrage, et à qui il demande d'afficher le contenu des Namespace Extensions qu'ils proposent. Ces plug-ins doivent être implémentés sous forme de composants COM...

2.2.2 Le modèle COM

Le modèle de programmation Objet tel qu'il est prôné par le C++ par exemple a apporté certains avantages du point de vue de la réutilisation des programmes par rapport au paradigme procédural du C ou du Pascal (pour ne citer que ces deux langages). Cependant, cette réutilisation restait au niveau du code source, à moins de disposer exactement du même compilateur et d'avoir compilé les composants à réutiliser sur le même système.

Microsoft a proposé une solution à cette limite en plaçant la réutilisation des composants au niveau du code exécutable et surtout en abolissant les frontières entre langages. Le nouveau modèle s'appelle COM : Component Object Model. Un objet COM est un objet au sens où l'entend le paradigme objet, et mis à part pour quelques spécificités, il s'utilise exactement comme n'importe quel objet du langage au sein duquel on l'utilise. Cependant, la différence réside dans le fait que le code exécutable du composant se trouve situé dans un *serveur* COM, qui peut être une bibliothèque DLL, ou même un autre exécutable ; par conséquent, le processus utilisant un objet COM est le *client*. Le procédé technique est complexe, et ce rapport n'a pas pour ambition de présenter COM avec ce niveau de détail. Voici toutefois une description sommaire de la manière dont fonctionne COM.

Lorsqu'on crée un objet COM, on commence par définir une ou plusieurs interfaces, c'est-à-dire des classes avec l'ensemble des méthodes qu'elles contiennent. Une telle interface se traduit par exemple en C++ par une classe virtuelle, c'est-à-dire où les méthodes n'ont aucune implémentation. Les interfaces de l'objet définissant ses capacités et son comportement, elles vont être publiée avec le code exécutable de l'objet, dans un langage particulier appelé IDL[†]. Le code de l'objet, lui, est écrit dans un langage au choix (dans la limite des langages proposés par Microsoft : C++ ou Visual Basic principalement).

On crée ainsi un couple Interface/Implémentation : pour une interface donnée, l'utilisateur peut ensuite choisir l'implémentation qu'il va utiliser. Il lui suffit pour cela de préciser au système quelle implémentation de l'objet il souhaite utiliser. Le système contient toutes les informations concernant cet objet, et notamment le serveur COM dans lequel il est implémenté. En effet, le modèle COM impose que les interfaces soient immuables. Une fois

publiée, une interface ne changera plus jamais. Ce qui permet à d'autres développeurs de proposer leur implémentation d'une interface précise.

Une fois qu'un objet COM a été créé, il peut être utilisé depuis plusieurs langages différents : C, C++, Visual Basic, Java ou encore C#.

Il est à noter que cette technologie est essentiellement réservée à Microsoft : il existe un modèle similaire, CORBA, utilisable sous UNIX et sous Windows, et très utilisé par Borland par exemple, mais COM ne fonctionne pour l'heure que sous Windows et est presque exclusivement supporté par des langages de chez Microsoft. Notons enfin que le nom COM est plus ou moins équivalent à OLE et ActiveX — ou du moins ces trois technologies sont-elles si fortement couplées qu'on peut les considérer comme équivalentes.

2.2.3 Les *Namespace Extensions*

Les Namespace Extensions sont ainsi des objets COM implémentant un certain nombre d'interfaces imposées par le Shell. Ainsi, le Shell sait comment utiliser les objets qu'il charge en tant que Namespace Extensions, car ceux-ci remplissent les conditions d'un contrat prédéterminé, c'est-à-dire qu'ils implémentes des interfaces bien connues.

Créer un Namespace Extension revient donc à implémenter des interfaces spécifiées par le Shell. Voici une petit liste non exhaustive des interfaces requises pour implémenter un Namespace Extension :

IShellFolder
: C'est la principale interface à implémenter : il s'agit de l'interface qui fait d'un objet un Folder du Namespace. Le Shell passe par cette interface pour demander des renseignements sur les objets (au sens du Shell) que contient le Folder.

IShellView
: Etant donné qu'il est possible de créer un Namespace Extension pour presque tout et n'importe quoi (il existe par exemple un Namespace Extension qui recrée l'arborescence de la Base de Registres directement dans le Shell, et un autre qui affiche une page Internet dans la partie droite de la fenêtre de l'explorateur, cf. figure 2), il est impossible à Explorer de savoir *comment* afficher les objets contenus dans un Folder. Il laisse donc ce travail à un objet implémentant l'interface IShellView. Un objet implémentant IShellView est responsable de l'affichage des données dans la partie droite de l'explorateur, celle qui d'habitude affiche les fichiers sous forme d'icônes, grandes ou petites, ou même sous forme de liste donnant des détails sur les fichiers. Ainsi, Explorer met à disposition la partie droite de sa fenêtre aux objets IShellView, seuls capables d'afficher convenablement le contenu de leur IShellFolder associé.

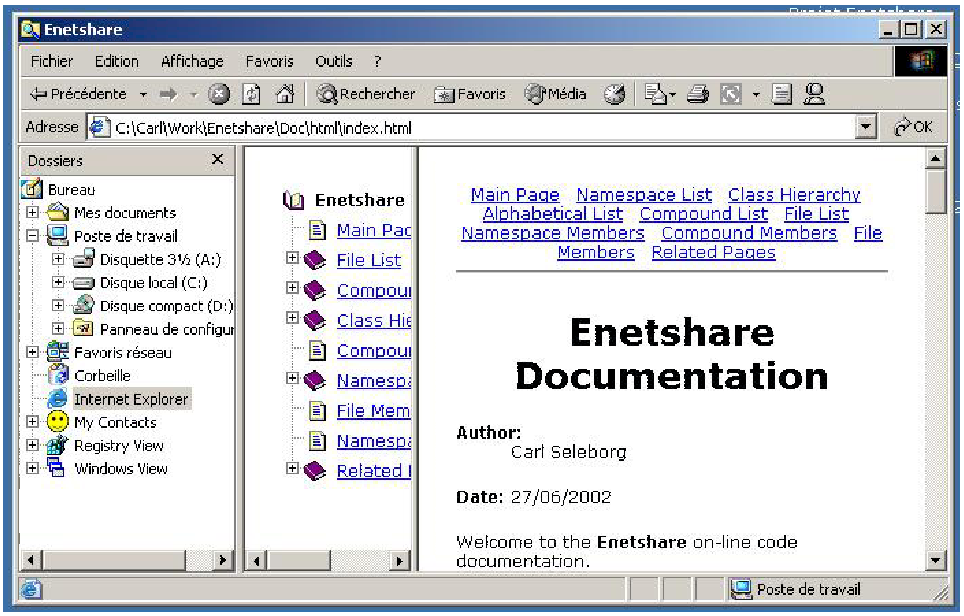

FIG. 2 – Le Namespace Extension d'Internet Explorer

IExtractIcon

Explorer affiche une icône pour chaque objet du Shell. Etant donné qu'il ne sait pas quelle icône choisir pour un objet précis (seul le créateur de cet objet sait quelle icône utiliser), le Shell propose l'interface IExtractIcon qui lui permet de récupérer l'icône d'un objet implémentant cette interface.

IDropSource, IDropTarget, IDataObject

Ces trois interfaces sont les interfaces-clés pour les opérations de Drag-n-Drop. Un objet IDropSource détermine le comportement visuel d'une opération de Drag-n-Drop. L'objet IDropTarget, lui, détermine le comportement du destinataire de l'opération de Drag-n-Drop. Enfin, IDataObject représente un objet encapsulant les données à transporter lors de l'opération de Drag-n-Drop. Cet objet sera passé de la source à la destination du Drag-n-Drop. C'est en inspectant cet objet que le destinataire détermine s'il est capable de gérer les données qu'on dépose sur lui.

2.2.4 Le serveur EnsShell

EnsShell est un serveur COM implémenté dans une DLL, qui contient tous les objets d'Enetshare relatifs à son intégration au Shell. Ces objets ne seront d'ailleurs pas uniquement utilisés par le Shell : l'application les utilisera aussi, afin d'assurer une bonne cohérence dans le comportement des

éléments visuels de l'application par rapport à ce que l'utilisateur verra dans le Shell. Plus particulièrement, on s'assurera ainsi que les menus contextuels (activés grâce au clic droit) sont bien les mêmes partout, que les objets réagissent de la même façon à une opération de Drag-n-Drop, etc... en utilisant systématiquement les objets COM proposés par EnsShell et destinés à remplir ces tâches.

Voici la liste des objets COM implémentés pour l'instant dans EnsShell. Pour chaque nom de classe donné, le nom affiché entre parenthèses indique le nom de l'interface COM qu'il implémente.

CFamilyShellFolder (IShellFolder)
: est l'objet qui représente aux yeux du Shell une famille d'éléments du Roster. C'est lui qui fournit au Shell tous les objets annexes dont ce dernier à besoin pour afficher chacun des éléments contenus dans cette famille, et surtout, c'est lui qui crée et renvoie au Shell des instances de CFamilyShellFolder qui représenteront les sous-familles (c'est ainsi que le Shell permet à l'utilisateur de "naviguer" dans les Namespace Extensions : chaque IShellFolder doit pouvoir renvoyer une nouvelle instance de IShellFolder représentant l'un de ses sous-folders). Ces objets sont décrits ci-après.

CFamilyShellView (IShellView)
: est l'objet qui affiche, dans la partie droite de la fenêtre d'Explorer, les contacts et les sous-familles contenues dans la famille sélectionnée. L'affichage, tout comme pour les fichiers, se fait au moyen d'une liste qui peut afficher les éléments sous diverses formes (grande icônes, petites icônes, détails, etc...).

CFamilyEnumIDList (IEnumIDList)
: est l'objet qui énumère tous les sous-objets contenus dans un CFamilyShellFolder. Le Shell utilise des PIDLs, des identifiants dont la structure est laissée à la charge du fournisseur du Namespace Extension, pour identifier chaque élément d'un IShellFolder. Donc, lorsqu'il souhaite avoir la liste des sous-éléments d'un IShellFolder, il demande à celui-ci de lui fournir un objet de type IEnumIDList, qui va lui énumérer les PIDLs de tous les sous-objets. Une fois muni de ces PIDLs, le Shell s'en servira pour demander au IShellFolder associé plus de détails sur chacun des éléments. Ainsi, le CFamilyEnumIDList énumère les PIDLs de tous les contacts et sous-familles appartenant à une famille associée à un CFamilyShellFolder.

CRosterObjectExtractIcon (IExtractIcon)
: se charge de fournir une icône pour un objet du Roster donné. Le Shell, une fois qu'il a le PIDL d'un élément d'un CFamilyShellFolder, demande à ce dernier un objet IExtractIcon qui lui fournira l'icône de cet élément. Un objet de type CRosterObjectExtractIcon sait parfaitement quelle icône associer à un élément précis du Roster.

CFamilyDropTarget (IDropTarget)
: est responsable de la réception d'un élément par opération de Drag-n-Drop. Plus précisément, on associe un élément graphique de Windows (comme une fenêtre ou un élément particulier d'une liste) avec un objet de type IDropTarget. Ce dernier, lorsque le Shell détecte qu'une opération de Drag-n-Drop va être effectuée sur l'élément graphique donné auparavant, est sollicité pour savoir d'une part s'il accepte l'opération de Drop (il doit pour cela déterminer les données qu'on lui apporte, celles-ci dépendant de la source du Drag-n-Drop) et d'autre part de gérer l'opération de Drop, c'est-à-dire d'incorporer convenablement les données du Drop. Un CFamilyDropTarget est également associé à une famille précise du Roster, et ne peut recevoir que des contacts ou d'autres familles (il ne peut recevoir de fichiers ou d'autres types de données). Lorsqu'on "droppe" des contacts ou des familles, celles-ci sont ajoutées à la famille associée.

CContactDropTarget (IDropTarget)
: est, quant à lui, responsable de la récéption d'un Drag-n-Drop sur une icône représentant un contact. A la différence des familles, les contacts ne peuvent, eux, pas recevoir d'autres contacts ou d'autres familles. En revanche, ils peuvent recevoir des fichiers, ce qui aura pour effet d'envoyer ces fichiers au contact représenté.

CFamilyListDropTarget (IDropTarget)
: utilise deux objets des deux classes précédentes pour gérer le Drag-n-Drop sur une liste d'objets du Roster. En fonction de la nature (contact ou famille) de l'élément sur lequel on passe avec la souris, il fera entrer en jeu l'un ou l'autre objet pour simuler le comportement correct.

CRosterObjectDataObject (IDataObject)
: est le moyen qu'utilise le Shell pour transférer des données via le Drag-n-Drop, mais aussi via le presse-papiers[†]. Un IDataObject encapsule les données à transporter, et peut les proposer dans différents formats, et via différents supports. C'est à la source du Drag-n-Drop de déterminer quels formats et quels supports le IDataObject envoyé doit proposer. Un object CRosterObjectDataObject contient, comme son nom l'indique, des objets du Roster, et est capable de les proposer sous forme de texte (en donnant le nom des objets, un nom sur chaque ligne), sous forme de fichiers (afin de pouvoir les conserver de manière persistante) et aussi sous un format propriétaire, connu du CFamilyDropTarget, afin de pouvoir gérer l'organisation du Roster par opérations de Drag-n-Drop.

CRODOEnumFormatETC (IEnumFORMATETC)
: est, à l'instar du IEnumIDList, un objet qui permet d'énumérer les formats proposés par un IDataObject. Le CRODOEnumFormatETC énumère les formats proposés par un CRosterObjectDataObject.

CBasicDropSource (IDropSource)
: est chargé de fournir le *feedback* visuel à l'utilisateur durant une opération de Drag-n-Drop. Principalement, cet objet modifie l'aspect du curseur selon que l'opération de Drag-n-Drop est une copie d'objets, un déplacement d'objets, ou si l'opération n'est pas autorisée. Comme son nom l'indique, cet objet est très simple.

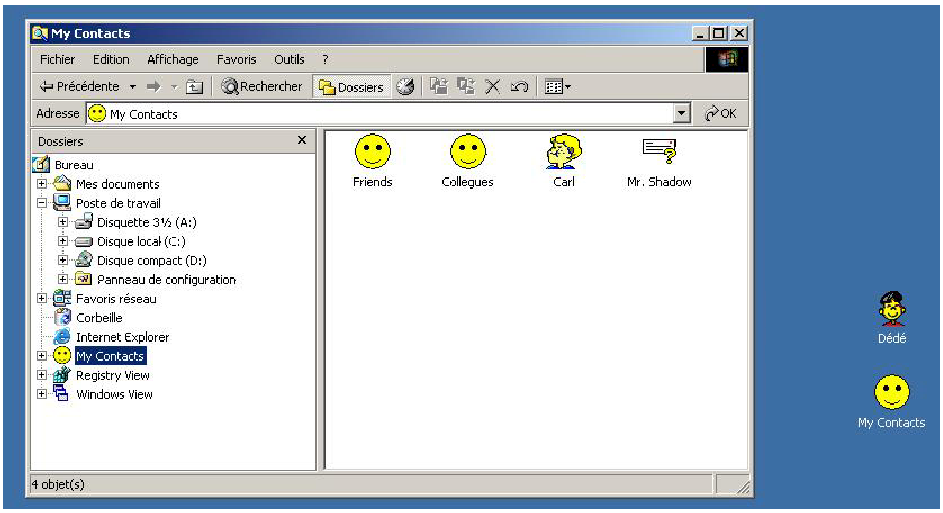

FIG. 3 – Enetshare dans le Shell

2.3 La boîte à outils EnsCommon

EnsCommon est le nom d'une DLL créée pour le projet Enetshare, destinée à rassembler un ensemble de classes et de fonctions communes à tous les *modules* du projet. En effet, il existe évidemment une *application* Enetshare, qui gère les évènements et les communications réseaux, et qui propose une partie de l'interface graphique du logiciel, mais il y a aussi la DLL EnsShell, qui ne fait pas partie de l'application Enetshare. Pour être précis, le code du module EnsShell sera en majorité exécuté par Explorer, ce qui signifie qu'il existe tout un ensemble de classes et de fonctions qui sont utilisés par deux programmes distincts. Afin d'éviter la duplication de ce code en mémoire, ces objets communs ont été placés dans une DLL à part : EnsCommon.

EnsCommon est une sorte de boîte à outils, contenant un certains nombre de classes (plus de vingt "vraies" classes, et plus de quarante si on compte toutes les petites classes assistantes), la plupart du temps assez générales, mais aussi quelques unes plus spécialisées pour une application bien précise, comme les classes qui composent le Roster (cf. paragraphe 2.3.1, p. 14). Voici une liste non-exhaustive des diverses outils que contient EnsCommon :

`CByteBuffer`
: Cette classe permet de manipuler un tableau d'octets en proposant des opérations d'écriture et de lecture séquentielles. Cette classe est très utilisée pour la sérialisation d'objets ainsi que pour la construction et la gestion des PIDLs.

Sérialisation
: Cf. paragraphe 2.3.3, p. 23.

Communication entre processus
: Cf. paragraphe 2.3.2, p. 20.

Gestion des erreurs
: Le logiciel Enetshare fonde toute sa gestion d'erreurs sur l'utilisation d'exceptions. EnsCommon propose une classe servant de base pour toutes les exceptions d'Enetshare, `EEnetshareException`, ainsi que des macros permettant d'utiliser ces exceptions en ayant plus d'informations de débogage.

Roster
: Cf. paragraphe 2.3.1.

2.3.1 Le *Roster* : le carnet d'adresses selon Enetshare

Le *Roster* est le carnet d'adresses de l'utilisateur Enetshare. Il s'agit d'une structure de données dans laquelle sont placés les *contacts* de l'utilisateur. Dans le code source, le Roster est représenté par la class `CRoster`, qui propose quelques services de base pour manipuler le Roster.

Les propriétés importantes du Roster. Le Roster possède quelques propriétés importantes qui ont conditionné son développement :

- Le Roster doit être géré de manière à ce que l'utilisateur puisse organiser ses contacts de manière logique. L'utilisateur doit pouvoir regrouper ses contacts en *familles*, et une famille doit pouvoir contenir des sous-familles.
- L'utilisateur doit pouvoir placer un contact dans deux familles distinctes, si jamais cela se justifie. Par exemple, une personne qui travaille avec son cousin pourrait avoir ce cousin dans la famille "Famille" et aussi dans la famille "Collègues". Si possible, l'utilisateur doit pouvoir donner des noms différents à un contact en fonction de la famille dans laquelle ce contact se trouve.
- Si un contact est présent dans deux familles différentes, toute modification doit être répercutée pour les deux exemplaires du contact. Ceci permet à l'utilisateur d'avoir l'impression qu'il n'existe véritablement qu'un "objet" contact représentant une personne et qu'il s'agit bien du *même* objet, quelque soit l'endroit où cet objet apparaît.

– Les informations contenues dans le Roster ne doivent pas être dupliquées, afin d'éviter tout problème d'incohérence : si il existe plusieurs instances du Roster, il devient difficile de répercuter tous les changements sur toutes les instances en même temps, en particulier si les différentes instances sont modifiées par des processus concurrents, comme cela peut être le cas sur un OS† multitâches comme Windows.

Ces propriétés ont servi de piliers à la conception d'un sous-système presque indépendant du reste de l'application, capable de gérer le Roster. Par extension, ce sous-système est également appelé *Roster*.

Les objets du Roster. Il convient de détailler un peu le contenu du Roster. La première propriété indique que les Familles sont des *conteneurs* d'objets qu'on appelle des Contacts. Ces contacts peuvent être de deux natures différentes suivant les informations qu'on possède sur la personne représentée.

Le logiciel Enetshare, comme la plupart des logiciels de messagerie modernes, retient toutes les adresses des personnes ayant envoyé des messages à l'utilisateur. Un peu comme pour les téléphones portables, ceci permet à quelqu'un de mémoriser ainsi l'adresse d'un exépéditeur, même s'il n'a pas explicitement entré les coordonnées complètes de cet expéditeur dans son Roster. Tout ce que l'utilisateur connaît de ce contact est alors son adresse e-mail, mais le contact ne porte pas de nom : il est *Anonyme*. Si un jour l'utilisateur décide de renseigner le Roster plus en détail sur cette personne, il devra au minimum fournir un nom et surtout, une icône à ce contact. Dès lors, cette personne devient un *Contact* dans le Roster Enetshare.

Ainsi, pour résumer, l'utilisateur peut avoir dans son Roster des Contacts, pour lesquels il a spécifié au moins une icône, et des Anonymes, qui sont des contacts dont il ne connaît que l'adresse e-mail.

Les Contacts, les Anonymes et les Familles sont les trois types d'objets qu'on peut trouver dans le Roster. En C++, cela se traduit par trois classes, `CRosterContact`, `CRosterAnonymous` et `CRosterFamily` qui dérivent toutes d'une base commune, `CRosterObject`. Chaque objet du Roster est identifié par un identifiant (cf. un peu plus loin), et presque toutes les opérations du Roster qui ont besoin d'identifier un objet précis dépendent d'un tel identifiant.

L'arborescence du Roster. Les trois premières propriétés rappellent celles du système de fichiers d'Unix, où le nom du fichier n'est pas lié directement au contenu, mais où le contenu est référencé par des liens qui existent dans l'arborescence des fichiers. Plus exactement, le contenu d'un objet du Roster est identifié par un GUID†. Les Familles, qui sont elles-même des objets du Roster, contiennent une liste de liens, chaque lien étant en réalité une pair nom/GUID. Ainsi, chaque lien renvoie à un autre objet du Roster

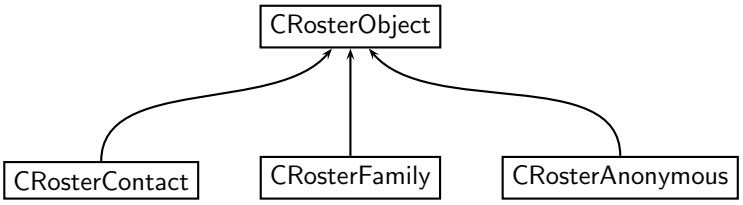

FIG. 4 – Diagramme d'héritage pour les classes d'objets du Roster

(celui dont le GUID correspond au GUID donné par le lien), et dans cette famille, le nom de cet objet est celui donné par le nom du lien.

Ce système de liens permet d'avoir un même objet à plusieurs endroits du Roster, et le fait de "nommer" le lien permet également d'avoir un nom différent pour chaque occurence de l'objet dans le Roster, la manière dont l'utilisateur souhaite identifier le contact pouvant changer selon le context. La figure 5 illustre ce cas.

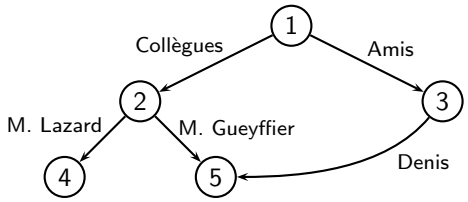

FIG. 5 – Denis Gueyffier est présent dans deux familles : Collègues et Amis

Dès lors que le Roster est présenté sous forme d'arborescence, dans laquelle aucune Famille ne peut contenir deux éléments portant le même nom, il est possible d'identifier une *occurence* d'un objet en donnant la suite de noms qui permet d'y arriver, exactement comme dans le cas des fichiers sous UNIX : `\Collègues\Recherche\Toto` est une occurence du Contact représentant une personne dénommée Toto et est classée dans la sous-Famille Recherche de la Famille Collègues, elle-même située à la racine du Roster. Ce

type de chemin est appelé FQN, ou *Fully Qualified Name*, soit Nom Pleinement Qualifié, dans l'application Enetshare (cf. figure 6). Il s'agit d'un moyen alternatif d'identifier des objets, et surtout, il s'agit du seul moyen d'identifier une *occurence* d'un objet. Si on souhaite renommer l'objet dans cette famille, ou si l'on désire supprimer cette occurence, il faut alors donner le FQN de l'objet, le seul identifiant de l'objet ne suffisant plus. Le Roster prévoit une opération de *résolution* de FQN qui renvoie le GUID de l'objet identifié par le FQN donné.

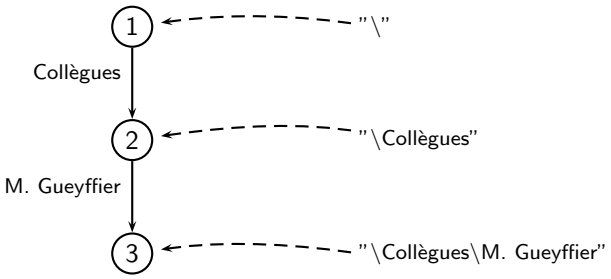

FIG. 6 – Exemples de FQNs et leurs objets associés

Une structure de données efficace. Le Roster a été prévu pour manipuler rapidement les objets qu'il détient. Pour atteindre ce but, et étant donné que lors des manipulations on ne donne pas l'objet en lui-même, mais plutôt son GUID, il a fallu créer une structure de données qui permette une association efficace entre un GUID et l'objet désigné par ce GUID. Il a donc fallu créer une class `CRosterObjectMap`, un conteneur associatif utilisant une table de hachage pour obtenir très rapidement un pointeur vers un objet du Roster à partir de son GUID. La recherche d'un objet étant donné son GUID, c'est la vertu des tables de hachages, est de l'ordre de $O(1)$, c'est-à-dire qu'elle est presque constante, et ne dépend (en théorie) pas du nombre d'objets contenus dans le Roster.

Les bits du GUID sont combinés ensemble grâce à deux opérations, la multiplication modulo 2^{32} et le *ou-exclusif* pour créer une valeur de hachage de 32 bits. Cette valeur, modulo la taille de la table de hachage, donne la rangée de la table à laquelle doit être inséré l'objet. Chaque rangée de la table de hachage est un tableau de paires GUID/pointeur, et le nouvel objet est inséré en fin de rangée. Pour une fonction de hachage parfaite, il n'y a aucune collision, c'est-à-dire qu'aucun objet n'est situé dans la même rangée qu'un autre. La fonction de hachage choisie n'est pas parfaite, mais affiche des résultats empiriques satisfaisants, en particulier lorsque la table

est dynamiquement redimensionnée pour ne jamais contenir plus de $0.75 \times$ *le nombre de rangées* éléments.

Sauvegarde du Roster. Le Roster est sauvegardé dans un fichier entre deux utilisations d'Enetshare. Ce fichier est en réalité un fichier XML[†], dont la syntaxe est relativement simple. Le XML a l'avantage d'offrir beaucoup de souplesse dans la manière de structurer les données, et c'est un format de plus en plus utilisé pour les applications liés au réseau et à l'Internet. Le fichier suivant illustre bien ce que sont les différents objets du Roster, ce qu'ils contiennent et quels sont les liens entre eux.

```
<Family ObjectID="0efb6b6b-b332-42d9-888b-0703e218d6c3">
   <Entry name="Friends" ObjectID="b31e609f-ed23-4dee-b5f1-eca2c2e2ca1e"/>
   <Entry name="Collegues" ObjectID="835ccc4b-f00f-4b3d-a67b-5a55c101b527"/>
   <Entry name="Carl" ObjectID="1d4d9f4c-4f2d-4978-b197-b65b572dd09b"/>
   <Entry name="Mr. Shadow" ObjectID="abba7a09-aba2-4ea3-be84-8db8d79424b2"/>
</Family>

<Contact ObjectID="1d4d9f4c-4f2d-4978-b197-b65b572dd09b">
   <Jid>carl.seleborg@enetshare.com</Jid>
   <FullName>Carl Seleborg</FullName>
   <IconId>102</IconId>
</Contact>

<Contact ObjectID="63cfe605-6283-4568-9d01-0e840228ad36">
   <Jid>denis.gueyffier@enetshare.com</Jid>
   <FullName>Denis Gueyffier</FullName>
   <IconId>101</IconId>
</Contact>

<Contact ObjectID="b9739035-ede5-446f-9ece-9ffb2abac29b">
   <Jid>julien.stern@cryptolog.fr</Jid>
   <FullName>Julien Stern</FullName>
   <IconId>104</IconId>
</Contact>

<Contact ObjectID="3ceb1f9a-8f92-4f0e-9882-868890b56d32">
   <Jid>baboonette@netcourrier.fr</Jid>
   <FullName>Elizabeth Ravier</FullName>
   <IconId>103</IconId>
</Contact>

<Anonymous ObjectID="abba7a09-aba2-4ea3-be84-8db8d79424b2">
</Anonymous>

<Family ObjectID="b31e609f-ed23-4dee-b5f1-eca2c2e2ca1e">
   <Entry Name="Babz" ObjectID="3ceb1f9a-8f92-4f0e-9882-868890b56d32"/>
   <Entry Name="Dédé" ObjectID="63cfe605-6283-4568-9d01-0e840228ad36"/>
   <Entry Name="Work Buddies" ObjectID="835ccc4b-f00f-4b3d-a67b-5a55c101b527"/>
   <Entry name="Root" ObjectID="0efb6b6b-b332-42d9-888b-0703e218d6c3"/>
</Family>



<Family ObjectID="835ccc4b-f00f-4b3d-a67b-5a55c101b527">
   <Entry Name="Julien Stern" ObjectID="b9739035-ede5-446f-9ece-9ffb2abac29b"/>
```

```
   <Entry Name="M. Gueyffier" ObjectID="63cfe605-6283-4568-9d01-0e840228ad36"/>
</Family>

<Contact ObjectID="a44c3140-5c0c-4b4a-ac87-35cd491b7f9f">
   <Jid>no.name@nowhere.com</Jid>
   <FullName>Mr. E</FullName>
   <IconId>105</IconId>
</Contact>
```

Cet exemple montre un Roster contenant 9 objets, le type de l'objet étant identifié par le nom du *tag* XML qui le définit : `Contact`, `Family` ou `Anonymous`.

La famille identifiée par le numéro 0efb6b6b-b332-42d9-888b-0703e218d6c3 est particulière : en effet, il s'agit du numéro de la famille *racine* du Roster (ce numéro est fixé, et sera toujours le numéro de la racine du Roster, quelque soit l'ordinateur sur lequel est installé Enetshare). Le Roster sait que lorsqu'il doit chercher la racine, il doit chercher ce numéro précis. Un fichier XML ne contenant pas de famille avec ce numéro est pour l'instant considéré comme invalide, car cela voudrait dire qu'aucun objet présent dans le fichier ne peut être affiché (puisqu'il n'y a aucun lien qui part de la racine, ce qui signifie que la racine est *vide*).

Notons au passage que l'objet a44c3140-5c0c-4b4a-ac87-35cd491b7f9f, le dernier contact du fichier, n'est pas atteignable, car il n'est référencé dans aucune famille. Le Roster prévoit ce cas et peut construire sur demande une famille spéciale qui contient les références de tous les objets (qui ne sont pas des familles, c'est-à-dire soit des contacts, soit des anonymes) qui ne sont pas atteignables en navigant depuis la racine. Il s'avère difficile d'empêcher l'utilisateur une manipulation malheureuse qui rendrait un ou plusieurs objets non-accessibles. Grâce à cette famille spéciale, il devient possible d'afficher à l'utilisateur la liste des contacts "perdus" afin qu'il les replace correctement dans le Roster. L'algorithme utilisé pour construire cette famille est proche de l'algorithme de Moore-Dijkstra tel qu'il a été vu en IUP 2 (cours de Recherche Opérationnelle).

Utilisation locale ou externe du Roster. La dernière propriété, qui veut que les données ne soient pas dupliquées implique que le Namespace Extension (c'est-à-dire la DLL EnsShell) communique avec l'application, qui est le seul processus instanciant le Roster (aucun autre processus n'a le droit de construire le même Roster en mémoire). Etant donné que c'est l'application qui détient le Roster mais que c'est dans Explorer que l'utilisateur effectue les manipulation, il est vite devenu nécessaire de ne pas seulement transmettre des données entre les deux processus, mais aussi des *instructions*, afin de dire "L'objet x a été déplacé dans la famille y" ou "Un lien vers l'objet x a été créé dans telle famille y et porte le nom z". Lorsque l'utilisateur effectue une manipulation dans le Roster, par exemple à l'aide du Drag-n-Drop, EnsShell envoie l'instruction correspondante à l'applica-

tion Enetshare, qui modifie le Roster, après quoi l'application peut renvoyer le nouvel état du Roster à EnsShell.

Ce mode de fonctionnement comporte deux points importants : la communication entre les processus doit être suffisament souple pour permettre l'envoi de divers types de données (celle-ci est détaillée au paragraphe 2.3.2, p. 20), et surtout, l'utilisation du Roster (pour le programmeur) doit être la même qu'il soit dans l'application Enetshare même ou dans un autre processus (autrement dit la programmation doit être la même, qu'il soit nécessaire ou non de passer par un mécanisme de communication entre processus, ou *IPC*[†]). En effet, une utilisation transparente du Roster, affranchissant le développeur de la communication entre processus, permet un développement plus rapide et plus indépendant du module dans lequel on travaille.

Cette transparence est obtenue grâce à l'utilisation d'une hiérarchie de classes présentée ci-dessous :

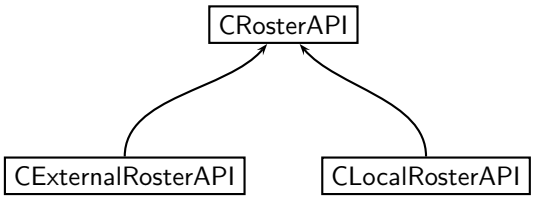

FIG. 7 – Diagramme d'héritage de CRosterAPI

L'interface commune `CRosterAPI` est le principal ingrédient dans l'utilisation identique du Roster quelque soit le processus. Il suffit simplement de choisir une implémentation de cette interface pour utiliser ou non la communication IPC. La classe `CExternalRosterAPI` implémente cette interface en utilisation la communication IPC, alors que la classe `CLocalRosterAPI` manipule directement l'appel du Roster par appels de méthodes de la classe `CRoster`. Mis à part le coût dû à la communication IPC, les deux implémentations sont strictement équivalentes, produisent les mêmes résultats et génèrent les mêmes erreurs (exceptions) en cas d'échec.

2.3.2 Communications entre processus

Etant donné que le projet Enetshare s'étendait sur au moins deux processus qui devaient collaborer étroitement en se communicant des informations et des instructions (l'application Enetshare et EnsShell), il était important

de créer un mécanisme de communication qui simplifie la programmation en offrant beaucoup de souplesse au développeur, afin qu'il puisse envoyer toutes les informations nécessaires. La souplesse était importante à partir du moment où, au stade ou en était le projet, tous les protocoles n'avaient pas encore été mis en place, et il fallait prévoir des ajouts ultérieurs, et même des modifications.

L'objet `CMessage`. L'unité de base de transmission de l'information est le `CMessage`. Il s'agit d'une sorte de conteneur associatif, couplant une clé sous forme de chaîne de caractères avec des données de divers types. Voici un exemple de `CMessage` contenant un entier nommé valeur, une chaîne nommée nom et une valeur de vérité (un booléen) nommée important :

```
CMessage message;
message.AddInt("valeur", 12);
message.AddString("nom", "Colis");
message.AddBool("important", false);
```

On peut ainsi véhiculer plusieurs valeurs, et de plusieurs types différents. Grâce à la sérialisation d'objets (cf. paragraphe 2.3.3, p. 23), il est même possible d'y inclure des objets sérialisés, ce qui implique qu'on peut y inclure d'autres messages, les objets `CMessage` étant eux-même serialisables :

```
CMessage conteneur;
conteneur.AddSerializedObject("message", message.Serialize());
```

La boîte à messages. Lorsqu'on envoie un message, il faut un destinataire. La class `CMessageBox` joue le rôle de ce destinataire. Le but était le suivant : créer un destinataire qui soit unique et visible à travers tout le système. Comme beaucoup d'objets du système Win32, les `CMessageBox` sont nommés, et le nom d'un objet `CMessageBox` doit être unique à travers tout le système. Grâce à ce nom, défini par avance la plupart du temps, un processus peut retrouver la boîte à messages d'un autre processus, et ainsi lui envoyer un message. Par exemple, si un processus a créé une boîte s'appellant MaBoite, on peut lui envoyer un message de la façon suivante :

```
CMessage message;
message.AddString("instruction", "quitter");
message.SendTo("MaBoite");
```

Réception du message. Lors de la récéption du message, la classe `CMessageBox` utilise une autre classe, `CMessageReceiver`, pour prévenir de la réception d'un message. Cette seconde class est en réalité une interface qui ne possède qu'une méthode, comme le montre la définition suivante :

```
class CMessageReceiver
{
public:
   void MessageReceived(CMessage& message, CMessage& reply) = 0;
};
```

Une classe qui souhaite recevoir des messages doit alors dériver de cette interface, implémenter la méthode `MessageReceived()` et se déclarer à la `CMessageBox` comme récepteur pour pouvoir réagir à l'arrivée d'un message. Plusieur récepteurs peuvent recevoir les messages arrivant à une `CMessage-Box`.

Dans la méthode `MessageReceived`, le premier argument est le message reçu, le second est un message vierge créé par la `CMessageBox` et qui sera renvoyé à l'expéditeur du message une fois que tous les récepteurs auront été informés du message. De cette manière, chaque récepteur peut mettre sa réponse dans le message `reply`.

Fonctionnement interne. En interne, la boîte de message est implémentée grâce à une fenêtre invisible, dont le nom est celui de la boîte de message. Windows permet d'énumérer les fenêtres présentes dans le système, ce qui permet de trouver le *Handle* de la fenêtre (son identifiant). Lorsqu'on envoie un message à une `CMessageBox`, on commence par sérialiser le message, puis on envoie les octets à la fenêtre grâce au message WM_COPYDATA. Lorsque la fenêtre reçoit ce message, elle effectue les trois opérations suivantes :

- Elle désérialise le message,
- Elle passe ce message à tous les récepteurs enregistrés, avec un second message de réponse, vierge celui-ci.
- Elle renvoie la réponse à l'exépéditeur.

Le renvoi de la réponse n'est pas aussi simple que l'envoi du message d'origine : en effet, si l'utilisation du message WM_COPYDATA, relativement simple, est possible, c'est bien parce que la boîte de messages est implémentée grâce à une fenêtre (seules les fenêtres peuvent recevoir les messages de type WM_*). Mais l'expéditeur du message, lui, n'a pas forcément une fenêtre sous la main pour recevoir la réponse de la même manière. Il faut donc procéder autrement.

La solution retenue est celle qui utilise une zone de *mémoire partagée* au sens Windows. Il s'agit d'une partie du fichier d'échange de mémoire virtuelle, dans lequel des processus distincts peuvent lire ou écrire des données. Ainsi, le récepteur crée la zone de mémoire partagée, la remplit avec les données de la réponse sérialisée, et informe l'expéditeur que la réponse a été postée dans cette zone de mémoire. A ce moment-là, le récepteur lit la zone de mémoire partagée, désérialise la réponse, et peut dès lors réagir correctement en fonction de la réponse.

2.3.3 Sérialisation d'objets

Principe. La sérialisation d'objets, en programmation, est un procédé qui permet de représenter une instance précise d'un objet par une suite d'octets ; entendu que ces octets reflètent l'*état* de l'objet au moment de sa sérialisation, c'est-à-dire qu'ils sont les *données* de cet objet. L'idée est alors de pouvoir stocker ces octets dans un fichier, ou encore de les envoyer sur un réseau, afin de rendre un objet *persistant* ou encore *transportable*.

La sérialisation n'est pas toujours un procédé trivial. Etant donné qu'un objet sérialisé est, la plupart du temps, utilisé par un processus différent de celui qui a sérialisé l'objet, il n'est pas possible, par exemple, d'utiliser des pointeurs dans la représentation sérialisée, puisque les pointeurs sont propres à l'espace mémoire d'un processus, et que chaque processus à son propre espace mémoire, distinct des autres. Les pointeurs à l'intérieur d'un processus n'ont plus de sens à l'intérieur d'un autre processus.

Le plus important était de rendre facile les opérations de sérialisation et de désérialisation d'un objet, et surtout de la rendre uniforme pour tous les objets capables d'être sérialisés. C'est à cette fin qu'a été créée la classe `CSerializable`, une classe abstraite (ou interface) qui ne contient qu'une méthode `Serialize()` renvoyant un objet de type `CSerializedObject`. Ce dernier est un objet qui gère la mémoire requise pour stocker les octets issus de la sérialisation (allocation, désallocation et nombre d'octets).

Ainsi, tout objet qui peut être sérialisé devrait dériver de `CSerializable`, afin de pouvoir être utilisé de manière uniforme. De même, tout objet sérialisable devrait avoir un *constructeur de désérialisation*, qui prenne en argument un `CSerializedObject`.

Le code suivant montre un exemple, avec un objet `CExemple` sérialisable :

```
CExemple objet1;

// Opérations sur objet1
// qui modifient son état

CSerializedObject so = objet1.Serialize();

// ...

CExemple objet2(so);
```

A la fin de cet exemple, objet2 doit se trouver exactement dans le même état que objet1 lors de sa sérialisation.

Applications. Au sein d'Enetshare, la sérialisation est utilisée principalement pour la transmission d'objets de l'application vers EnsShell, ou inversement. Elle est également utilisée pour stocker un Contact sur le bureau, le

Contact étant stocké sous forme d'un fichier contenant les données sérialisées de ce Contact (ce qui implique que la classe `CRosterContact` est sérialisable. En réalité, c'est la classe `CRosterObject` qui est sérialisable, et par héritage, toutes les sous-classes le sont aussi).

2.4 Autres travaux

Mon travail ne s'est pas limité à la conception et la l'implémentation des classes présentes dans EnsShell, EnsCommon ou Enetshare. L'idée d'intégrer l'appliation au Shell n'est venue qu'après avoir fait divers essais différents de placer des icônes sur le bureau, et avant même de m'atteler à cette (vaste) tâche, il m'a été demandé d'autres travaux.

2.4.1 Le contrôle ActiveZone

Lorsque je suis arrivé, on avait déjà proposé une méthode pour afficher des contacts. Cette méthode se basait sur une interface légèrement différente de celle souhaitée finalement par M. Gueyffier, qui reprenait le principe de l'interface du logiciel ICQ, à savoir une fenêtre placée sur le bord de l'écran contenant une liste d'icônes de contacts. Ces icônes, alors qu'elles sont en général statiques dans le Shell, devaient réagir aux mouvements de la souris et au Drag-n-Drop. Ces icônes étaient en réalité des *contrôles* Windows, au même titre que les boutons ou même que les fenêtres.

Sous Windows, les contrôles sont des fenêtres (au sens Windows), c'est-à-dire qu'elles ont une procédure spéciale qui leur permet de recevoir les messages que le système lui envoie ; et ils sont nombreux : il y a des messages pour la souris, pour le clavier, pour la taille ou la position de la fenêtre, pour des évènements systèmes particuliers... la liste est longue (ce sont les messages `WM_*`).

Lorsqu'on crée un nouveau contrôle, la plus grosse partie du travail consiste à reprogrammer cette procédure de messages, afin de réagir convenablement aux divers évènements, et afin de se donner une apparence propre. Le contrôle *ActiveZone* est un contrôle spécialement créé pour Enetshare, dans le but d'avoir un élément graphique qui se comporte comme souhaité et qui soit réutilisable à volonté. A cette fin, il est implémenté sous forme de classe, avec un certain nombre de méthodes virtuelles qu'il faut réimplémenter dans une classe dérivée afin de définir son propre comportement, l'ActiveZone telle quel ne faisant pas grand-chose. Ses principales propriétés sont :

Image et forme
: Le contrôle ActiveZone n'a pas de représentation propre par défaut : il faut impérativement implémenter la méthode `Paint()` de la classe `CActiveZone` pour pouvoir donner une représentation graphique au contrôle. A l'origine, l'ActiveZone était faite pour afficher des images,

auquel cas on charge l'image lors de la création du contrôle, et on l'affiche dans la méthode `Paint()`. On peut également facilement spécifier une forme différente de sa forme rectangulaire de base, créant ainsi des zones "transparentes" à l'intérieur ou sur les bords du contrôle.

Mouvements de souris
L'ActiveZone est sensible aux quatre évènements de souris principaux, à savoir l'entrée du curseur dans le contrôle, la sortie du curseur, le déplacement du curseur à l'intérieur du contrôle et les actions sur les boutons de la souris.

Drag-n-Drop
L'ActiveZone peut recevoir des fichiers qu'on dépose dessus par opération de Drag-n-Drop.

Double-buffering
Un changement fréquent de l'affichage de l'ActiveZone peut engendrer un scintillement, du fait qu'on efface en général le dessin précédent avant de redessiner par-dessus. En demandant à l'ActiveZone d'utiliser le double-buffering (qui utilise plus de mémoire), on peut supprimer ce scintillement. Ce mécanisme ne nécessite aucune intervention du programmeur, car il est pris en charge dans la classe `CActiveZone` elle-même.

Placer des ActiveZones dans une fenêtre n'était pas difficile, et semblait même assez naturel. Mais lorsqu'il s'est agi de placer des icônes sur le bureau, et que nous avons essayer de le faire grâce aux ActiveZone, la solution s'est avérée gourmande en resources et au final peu viable, ce pourquoi nous avons ensuite opté pour l'intégration avec le Shell.

Cependant, l'ActiveZone n'est pas passée aux oubliettes pour autant. Etant donnée qu'elle avait été conçue pour être souple d'utilisation (et relativement simple aussi), elle sera sans doute utilisée pour créer les fenêtres de l'interface graphique d'Enetshare, notamment les fenêtres dans lesquelles l'utilisateur entrera les messages qu'il souhaite envoyer à ses correspondants. Le fait qu'on puisse facilement afficher des images ou lui donner une forme libre permettra sans doute d'obtenir rapidement des fenêtres originales, un peu à l'instar de l'assistant de Microsoft Office, cette petite figure en forme de trombone qui "vit" sur une petite feuille de papier.

2.4.2 MAPI, ou comment importer des données depuis Outlook

L'une des premières tâches qui me furent confiées, dès le début de stage, fut d'étudier la possibilité d'importer le carnet d'adresses de l'utilisateur depuis Microsoft Outlook, afin de lui faciliter la transition d'Outlook vers Enetshare en lui permettant de conserver toutes les informations de son carnet d'adresses. Après avoir fouillé dans la documentation de Microsoft, il est vite devenu évident qu'il fallait passer par MAPI[†], ou *Messaging API*,

l'API du système qui gère la plupart des tâches et des données liées au courrier électronique.

MAPI, je m'en suis rendu compte après coup, est basé sur COM (ce qui explique sans doute mes difficultés à comprendre son fonctionnement au départ). Il s'agit d'un ensemble d'interfaces (et de classes implémententes) permettant de travailler de manière très intime avec Outlook, ou n'importe quel autre logiciel de mail compatible MAPI (relativisons : bien que la norme soit ouverte et documentée, étant donné qu'il s'agit là d'une API créée par Microsoft, Outlook est le seul logiciel à l'implémenter entièrement, avec Lotus Notes).

Grâce à ces classes et fonctions MAPI, il était donc possible de lire toute l'arborescence du carnet d'adresses Outlook de l'utilisateur et d'en extraire les informations appropriées pour les sauvegarder sous un autre format, lisible par Enetshare. Pour illustrer cela, j'ai créé une petite application (assez simple), capable d'extraire *la plupart* des informations du carnet d'adesses Outlook et de les afficher à l'écran. Celui qui sera chargé de compléter cette tâche pourra trouver dans cette application un bon exemple de comment procéder.

2.4.3 Doxygen, ou la génération automatique de documentation

On ne programme jamais sans écrire de documentation. Partant de ce sacro-saint principe, je me suis très tôt lancé dans l'écriture d'une documentation de l'ActiveZone et de tout mon travail du début du stage. Mais la documentation était écrite séparément, et n'était aucunement liée au code, ce qui impliquait qu'il fallait écrire deux fois ce que faisait chaque classe et chaque fonction : une fois dans le code, dans les commentaires, et une fois dans la documentation. Entreprise fastidieuse, car non seulement faut-il écrire deux fois la même chose, mais il faut aussi corriger les erreurs et effectuer les mises à jour à deux endroits différents, ce qui représente une perte de temps considérable.

Lorsqu'on programme en Java, il existe un outil, JavaDoc, qui génère une documentation HTML à partir des commentaires du code source. L'outil s'occupe de créer automatiquement les références qu'il faut, ce qui donne au final une documentation navigable, où l'on peut cliquer sur le nom d'une classe pour aller directement à la page où celle-ci est documentée. Après quelques petites recherches, j'ai trouvé un utilitaire similaire, Doxygen, qui fait pratiquement la même chose pour le code C++ (entre autres). J'ai donc assez vite choisi d'utiliser cet outil pour créer la documentation d'Enetshare, et j'ai ainsi pu documenter toutes les classes et fonctions des modules EnsCommon, EnsShell et Enetshare, sans redondance et avec des mises à jour très simples et fréquentes. La documentation contient également des pages décrivant certaines parties du système plus en détail, comme par exemple le fonctionnement du Roster ou la trasmission de messages IPC, ainsi que

des notes que j'ai laissées pour guider les développeurs qui reprendront mon travail par la suite.

3 Le déroulement du stage

Cette section décrit le déroulement général du stage, présente les personnes avec lesquelles j'ai eu l'occasion de travailler et les conditions dans lesquelles s'est effectué le stage.

3.1 Les intervenants

Enetshare n'est pas une grande entreprise. Durant le stage, j'avais pris l'habitude de présenter la chose en disant : "*Il y a le patron, et puis il y a le stagiaire*". C'est une entreprise sans salarié, et cela limite forcément la taille de l'équipe au sein de laquelle j'ai travaillé. Je dirais que la plupart du temps, j'ai travaillé seul, sauf au début et à la fin.

Lorsque je suis arrivé, déjà quatre personnes travaillaient sur le projet. Il y avait bien sûr M. Gueyffier, mais aussi trois personnes intervenant sur le projet en qualité de consultants dans différents domaines.

Celui qui a travaillé le plus sur ce projet, et qui s'est surtout occupé de la partie réseau, s'appelle **Vincent Ribailler**. Etudiant en dernière année de l'école des Télécoms, spécialisé dans les réseaux, il s'est vu confier la tâche de trouver une bonne architecture de classes pour l'application, et a fait des essais avec la bibliothèque JabberCOM, qui propose des interfaces COM pour communiquer avec Jabber, "l'épine dorsale" du système Enetshare.

Julien Stern travaille pour une entreprise qui s'appelle Cryptolog, une société de conseils en matière de sécurité informatique et cryptographie. Julien Stern est l'un des grands experts français en matière de cryptographie, et il a apporté au projet son expertise en la matière pour assurer une bonne sécurité des transmissions de données. En effet, Enetshare est un logiciel destiné (pour le moment) principalement aux professionnels, ce qui implique qu'il doit pouvoir satisfaire toutes les demandes en matières de sécurité et de confidentialité, et de ce point de vue, avoir le logo de Cryptolog apposé sur le produit est véritablement un atout. Bien que je n'ai vu M. Stern qu'une seule fois, ce fut fort intéressant, la cryptographie étant un domaine qui m'intéresse beaucoup. Malheureusement, bien que je l'aurais souhaité, je n'ai pas pu m'impliquer dans cette partie-là du logiciel, étant donné qu'il s'agit d'un domaine bien moins permissif et tolérant aux erreurs que d'autres (comme les interfaces graphiques, par exemple).

Enfin, celui avec lequel j'ai eu le plus de contacts sur le plan du travail fut **Franck Valetas**, ingénieur chez Mapstan, une entreprise dont les produit se fonde sur un algorithme de *cartographie de données* sur Internet. Franck était chargé d'apporter des idées concernant l'interface graphique et d'autres points liés à Windows et à COM, lui-même ayant une grande expérience en la matière. C'est lui qui nous a mis sur la piste MAPI (cf. paragraphe 2.4.2, p. 25) et c'est lui qui a fait les premières expériences avec les ActiveZones.

Ces trois personnes n'ont pas travaillé très longtemps avec Enetshare

après mon arrivée dans l'entreprise, si bien que je me suis très vite retrouvé seul à programmer, M. Gueyffier étant (très) occupé par la partie administrative et financière de sa *start-up*. Durant environ deux mois, les choses furent ainsi et, mis à part quelques mails envoyés à Franck Valetas avec quelques questions, j'ai du me débrouiller seul la plupart du temps pour résoudre les nombreux problèmes qui s'offraient à moi. Ce n'est que vers le mois d'août, c'est-à-dire le dernier mois du stage, que M. Gueyffier a pu se consacrer à la programmation, principalement sur la partie réseaux.

3.2 La phase d'apprentissage

Dire que le stage fut instructif serait un euphémisme : il ne s'est pas passé un seul jour sans que je consacre au moins une heure à lire la documentation technique de Microsoft. Si obtenir des détails sur une fonction particulière de l'API Win32 est une chose, comprendre les mécanismes de COM en est une autre, qui requiert une longue phase d'apprentissage avant de pouvoir en faire quelque chose.

C'est ainsi que j'ai passé près de trois semaines à me documenter sur COM, soit dans des livres, soit sur Internet, soit dans la MSDN. Il faut avouer que ce modèle remet beacoup d'automatismes en cause, et qu'il impose de changer radicalement sa conception de la programmation objet. Le point noir de cette technologie, c'est qu'il faut souvent fournir beaucoup de travail avant de pouvoir observer le moindre résultat. La preuve la plus flagrante en est le fait qu'il soit nécessaire de passer par les *Wizards* de Visual C++ pour créer un serveur COM vide : sans même avoir écrit la moindre ligne de code, les sources comprennent déjà plus de 400 lignes de C++.

A côté de cela, apprendre à utiliser l'API Win32 s'est avéré beaucoup plus simple, bien qu'infiniment plus long : alors qu'avec COM, l'important est de comprendre le *principe*, l'API Win32, elle, n'est qu'un ensemble de fonctions, structures, types et objets dont il faut connaître les *détails*, ce qui justifie d'avoir toujours la documentation sous la main lorsqu'on programme une application Windows. Mon sentiment personnel sur cette API est que c'est probablement l'une des plus laides et les moins bien organisées qui soit, mais elle a le mérite d'être finalement assez simple (bien que laborieuse) et surtout, de produire des résultats. Elles est devenue si vaste avec les années que presque toute sortes de fonctionnalités y sont présentes, et de ce point de vue, elle est très riche.

MAPI fut également un défi en matière d'apprentissage, d'autant plus qu'il s'agit de COM caché : ça fonctionne comme du COM, mais personne ne le dit nulle part. Je n'ai (heureusement ?) pas passé plus de deux semaines sur MAPI, étant donné qu'il ne s'agissait là que d'un travail très spécifique qui était la récupération du carnet d'adresses d'Outlook en vue de le réimporter dans Enetshare.

Enfin, à mis-chemin entre COM et l'API Win32, on trouve l'API du

Shell de Windows. Là encore, rien de très difficile à comprendre, si ce n'est qu'il faut beaucoup tâtonner avant d'arriver au moindre résultat utile.

Visual C++ et, plus généralement, le C++, n'ont pas présenté de soucis particuliers, étant donné que c'étaient là des outils que je connaissais déjà.

3.3 Les conditions de travail

Durant ce stage, j'ai eu à ma disposition la plupart des outils nécessaires à mon travail. Pour ce qui est de la configuration matérielle, j'avais un ordinateur puissant (Athlon 1GHz et pléthore de mémoire vive) et un écran agréable (élément indispensable au bien-être du développeur). J'ai essentiellement travaillé sous Visual C++, documentation incluse, et les quelques rares moments où j'avais besoin d'autres outils, je pouvais en général en trouver une version gratuite sur Internet, téléchargeable rapidement grâce à la connexion ADSL.

Enfin, en matière de bibliothèque, j'avais à ma disposition quatre livres essentiels :

- *The C++ Programming Language*, de Bjarne Stroustrup, le livre de référence du langage C++,
- *Le Guide de l'API Win32*, un livre reprenant en détail toutes les fonctions et structures de ladite API,
- *La Bible PC - Programmation Système*, un de mes propres livres que j'ai pris soin d'amener avec moi au bureau, et enfin
- *Programming the Windows Shell* de Dino Esposito, un livre au format PDF qui explique en détail la programmation du Shell sous tous ses aspects.

Quant au café, il était disponible à volonté.

3.4 La fin du stage

J'ai consacré le derniers jours du stage à préparer ma succession. J'ai arrêté la programmation une semaine avant la fin, pour augmenter et finir la documentation du code, afin que le suivant puisse reprendre le travail là où je l'ai laissé.

Conclusion

Mon stage chez Enetshare s'est dans l'ensemble très bien passé, et j'ai vraiment le sentiment d'avoir appris de nouvelles choses : bien que je connaissait déjà un peu la programmation Windows, j'ai découvert COM, ATL[†], le Shell, MAPI et bien d'autres technologies dont j'ignorais presque l'existence, et auquelles je n'aurais probablement pas été confronté avant longtemps, car elles ne sont pas au programme de l'IUP. C'est dommage, car maintenant que j'ai vu ces domaines, je me sens réellement capable de dire : "je peux programmer une application Windows", ce qui n'est pas négligeable sur un marché de l'informatique dominé par Microsoft.

Pour ce qui est de l'enseignement de l'IUP, c'est surtout la partie du Génie Logiciel, et en particulier l'UML qui m'a été utile, avec également un soupçon de recherche opérationnelle — je constate aujourd'hui que je n'avais pas hésité une seconde sur la manière d'effectuer la recherche des éléments perdus du Roster (cf. p. 19), tâche pour laquelle je me suis servi de l'algorithme de Moore-Dijkstra vu en cours en IUP 2. Par contre, le reste de mes connaissances, et notamment le C++, je les avais déjà acquises par moi-même, avant même d'entrer à l'IUP.

J'ai juste un regret concernant mon travail. Etant donné que je ne connaissais pas bien les domaines dans lesquels je m'aventurais, j'ai effectué la majorité de la conception *en même temps* que la programmation, ce qui m'a valu de devoir recommencer certaines choses avant d'en être satisfait. Une phase de conception plus longue, et avec l'aide de quelqu'un connaissant déjà bien la programmation Windows aurait sans doute été plus efficace. J'ai aussi parfois trouvé que nous avions, M. Gueyffier et moi-même, un peu de mal à coordonner nos travaux sur un plan technique, et j'ai plusieurs fois constaté que lorsqu'il utilisait du code que j'avais écrit, il n'utilisait pas les bonnes classes, ou il les utilisait mal ; sans doute ma documentation manquait-elle de précision ou était-elle trop éparpillée pour servir convenablement le reste de mon travail.

C'est mon second stage dans une très petite entreprise, et c'est sans doute pour cela que c'est aussi la seconde fois qu'on me confie non seulement un travail intéressant, mais surtout un travail sur lequel repose l'avenir de l'entreprise. C'est très flatteur de savoir qu'on m'a fait confiance à ce point. Par contre, je pense maintenant qu'il sera aussi utile d'effectuer mon troisième (et dernier ?) stage dans une structure plus large, dans laquelle je pourrais travailler au sein d'une équipe, avec des personnes compétentes qui m'apporteront sans doute plus que ne peut m'apporter la documentation technique ou les livres.

Quoi qu'il en soit, j'espère — et je pense — avoir fait un bon travail, et je souhaite vivement qu'il soit repris par d'autres développeurs pour en faire un logiciel complet qui, à mon avis, saura, par son originalité, trouver un écho auprès des utilisateurs.

Annexe — Un exemple de code source

Le code source suivant définit la classe `Enetshare : :CObjectID` qui est une encapsulation d'un GUID, facilitant l'utilisation de ces identifiants dans l'application Enetshare. Il montre également la syntaxe des commentaires destinés à Doxygen, l'outil de documentation utilisé pour générer la documentation du projet.

```
/**
* Defines CObjectID.
*
* Subdirectory : EnsCommon.
*
* @author Carl Seleborg
* @date 07/08/2002
* @file
**/

#ifndef _OBJECTID_H_
#define _OBJECTID_H_

#ifndef _ENSCOMMON_H_
#error You should include "EnsCommon.h" instead of "ObjectID.h"
#endif

#include "EnsDefines.h"
#include "Serializable.h"
#include "Message.h"

namespace Enetshare
{
    /**
    * The CObjectID class represents the general concept of a unique object
    * identifier. The implementation uses a GUID structure as object identifier,
    * and the class provides some basic services for handling and managing
    * such identifiers.
    *
    * ObjecIDs can be ordered. Operators are provided to order them by
    * their byte values.
    **/
    class ENSCOMMON_API CObjectID : public CSerializable, public CMessagePackable
    {
    public:
        /**
        * Default constructor. Creates a new random identifier.
        **/
        CObjectID();

        /**
        * Constructor. Initializes the identifier with the given GUID.
        **/
        CObjectID(const GUID& guid);

        /**
        * Parses a string representation of an ObjectID and intializes the object.
        * The string representation should have the same format as the one returned
        * from GetString().
        *
        * @param str String representation of the object identifier.
        *
```

```
* @throw EObjectIDParseFailed The string representation could not be
* parsed correcly and is probably invalid.
**/
CObjectID(const TString& str);

/**
* Copy constructor.
**/
CObjectID(const CObjectID& other);

/**
* Deserialization constructor.
**/
CObjectID(const CSerializedObject& serialized);

/**
* Message-unpacking.
**/
CObjectID(const CMessage& message);

/**
* Assignment (copy) operator.
**/
CObjectID& operator=(const CObjectID& other)
{
    if(this != &other)
        m_guid = other.m_guid;
    return *this;
}

/**
* Comparison operator.
**/
bool operator==(const CObjectID& other) const
{
    return memcmp(&m_guid, &other.m_guid, sizeof(GUID)) == 0;
}

/**
* Difference operator.
**/
bool operator!=(const CObjectID& other) const
{
    return !(*this == other);
}

/**
* Less-than operator.
**/
bool operator<(const CObjectID& other) const
{
    return memcmp(&m_guid, &other.m_guid, sizeof(GUID)) < 0;
}

/**
* Less-than-or-equal-to operator.
**/
bool operator<=(const CObjectID& other) const
{
    return memcmp(&m_guid, &other.m_guid, sizeof(GUID)) <= 0;
}
```

```
        /**
        * Greater-than operator.
        **/
        bool operator>(const CObjectID& other) const
        {
            return !(*this <= other);
        }

        /**
        * Greater-than-or-equal-to operator.
        **/
        bool operator>=(const CObjectID& other) const
        {
            return !(*this < other);
        }

        /**
        * Computes a 32-bit hash value of the identifier, to be used
        * in hash tables and other places where hashing is useful.
        *
        * @return A 32-bit hash value of the identifier.
        **/
        inline THashValue Hash() const;

        /**
        * Gets the size, in TCHARs, of the string representation of
        * this identifier, excluding the null-terminator.
        **/
        size_t GetStringSize() const { return 36; }

        /**
        * Gets the size, in bytes, of the actual identifying data.
        **/
        size_t GetSize() const { return sizeof(GUID); }

        /**
        * Returns a pointer to the actual identifying data.
        **/
        void* GetData() const { return (void*)&m_guid; }

        /**
        * Fills a TCHAR buffer with the string representation of this identifier.
        *
        * The string representation of an identifier is it's hexadecimal value,
        * using lowercase characters, with the format commonly used in the
        * Registry. An example of this is :
        *      "7a4fa547-82cc-4816-b348-2ea922a855a6"
        *
        * @param nMaxChars Size of the receiving buffer in TCHARs. If the string
        * representation is larger, it will be truncated to fit. If negative,
        * all characters will be copied.
        *
        * @return Pointer to pBuffer.
        **/
        TCHAR* GetString(TCHAR* pBuffer, size_t nMaxChars = -1) const;

        /**
        * Packs the ObjectID into a CMessage.
        **/
        void PackInto(CMessage& message) const;

#ifdef _DEBUG
```

```
        /**
        * Conversion to TCHAR*. This returns the address of an internal
        * buffer containing the string representation of this identifier.
        * Note that it's safer to use GetString().
        **/
        operator TCHAR*() const;
#endif // _DEBUG

        /**
        * Conversion to void*. This returns a pointer to the beginning
        * of the GUID structure inside.
        **/
        operator void*() const { return GetData(); }

        /**
        * Serializes the object.
        **/
        virtual CSerializedObject Serialize() const;

        /**
        * Creates and returns a new random CObjectID.
        **/
        static CObjectID Random(void);

        /**
        * Creates and returns a new zero CObjectID. This is useful
        * to tell if an object id has been initialized or not, much
        * like with pointers.
        **/
        static CObjectID Zero(void);


    private:
        /// Internal structure of the identifier.
        GUID m_guid;

        /// Serialization magic constant to check for data consistency
        static const long SERIALIZE_MAGIC;

#ifdef _DEBUG
        /**
        * Internal buffer used when returning a pointer to a string
        * representation of the identifier. This is a comodity, but
        * since the buffer is shared, it's safer to use the GetString()
        * method.
        **/
        static TCHAR m_s_stringBuffer[];
#endif


#ifdef _DEBUG
    public: static void _Test();
#endif
    };

    //---------------------------------------------------------------------------

    DECLARE_EXCEPTION(EObjectIDException, EEnetshareException, "General ObjectID exception");

    DECLARE_EXCEPTION(EGuidCreationFailed, EObjectIDException, "Failed to create a GUID");

    DECLARE_EXCEPTION(EObjectIDParseFailed, EObjectIDException, "Could not parse ObjectID");
```

```
    /**
    * Tells that an object id is already in use. It does not necessarily mean that
    * you generated an ObjectID that was identical to a previously generated one. This
    * is highly unlikely. However, it can mean that, in the context where you used
    * the ObjectID, you had already used it and that that situation is unacceptable.
    **/
    DECLARE_EXCEPTION(EDuplicateObjectID, EObjectIDException, "Duplicate ObjectID");
}

#endif // _OBJECTID_H_
```

Glossaire

ATL *(p. 31)* : L'*Active Template Library* est une bibliothèque de modèles créée par Microsoft pour faciliter la programmation et l'utilisation d'objets COM en C++.

COM *(p. 1)* : *Component Object Model* est un modèle objet conçu par Microsoft pour permettre de réaliser des objets réutilisables dans plusieurs langages. COM est aussi appelé OLE ou ActiveX.

DLL *(p. 1)* : *Dynamic Link Library* ou Bibliothèque de Liason Dynamique. Une DLL est une bibliothèque de fonctions et de classes qu'une application peut charger lors de son exécution (d'où son aspect *dynamique*). L'avantage des DLLs est que plusieurs applications peuvent ainsi partager du code qui lui, n'est chargé qu'une fois en mémoire.

Drag-n-Drop *(p. 6)* : Traduit en Français par Glisser-Déposer, le Drag-n-Drop est une opération visuelle par laquelle l'utilisateur déplace ou copie un objet du Shell d'un endroit à un autre, à l'aide de la souris. Le Drag-n-Drop ne fonctionne pas uniquement pour les fichiers et les répertoires, bien que ce soit là son utilisation la plus courante. On peut également, par exemple, effectuer un Drag-n-Drop pour créer un raccourci vers presque n'importe quel type d'objet du Shell.

GUID *(p. 15)* : Un *Globally Unique Identifier* est une valeur, codée sur 128 bits, et qui est générée par un algorithme complexe qui lui confère une propriété intéressante : deux GUIDs générés par cet algorithme sont, avec une très grande probabilité (presque sûrement), différents, et ce même s'ils sont générés par deux machines séparées au même moment, ou par une même machine à deux instants différents. Pour ce faire, l'algorithme utilise entre autres l'horloge interne de l'ordinateur ainsi que l'adresse IEEE de sa carte réseau, qui, elle, est unique au monde (l'unicité étant garantie par un organisme spécial, le *IEEE Registration Authority*). Dans le monde Windows/COM, on parle aussi souvent de IID, UUID ou encore CLSID, qui sont autant de synonymes de GUID. Un GUID est souvent représenté sous la forme suivante : {0a87d920-c57c-11d6-99cb-0004769a89b0}.

IDL *(p. 8)* : Le langage IDL, ou *Interface Definition Language* est un langage particulier destiné, comme son nom l'indique, à la création d'interfaces d'objets COM. La création d'un tel langage était nécessaire pour s'assurer qu'une interface serait compréhensible par plusieurs langages de programmation différents, étant donné que c'est l'un des principaux autouts de COM : le choix du langage pour utiliser un composant particulier.

IPC *(p. 20)* : Le terme *Inter-Process Communication* désigne le fait de communiquer entre processus distincts du système. Ce mode de communication est rendu difficile par le fait que deux processus ne partagent pas de mémoire en temps normal, étant donné que le mécanisme de mémoire virtuelle sépare clairement les espaces mémoire des processus. Les mécanismes d'IPC sont propres à l'OS sur lequel ils sont effectués ; Windows propose à cette fin les *Pipes*, les *Mailslots*, les messages de fenêtres classiques ou encore les *Zones de Mémoire Partagées*, ces dernières passant le plus souvent par l'utilisation d'un fichier).

MAPI *(p. 25)* : La *Messaging API* de Microsoft est une interface de programmation permettant aux développeur de bénéficier de toutes les fonctionnalités de Microsoft Outlook, ou de tout autre serveur MAPI, ce qui permet notamment de pouvoir envoyer ou recevoir des mails sans avoir à programmer quoique ce soit lié au réseau ou aux protocoles utilisés dans ce cas.

OS *(p. 15)* : *Operating System*, ou *Système d'Exploitation* en français. Il s'agit du "logiciel" qui contrôle et pilote l'ordinateur, et qui permet à l'utilisateur d'accéder aux divers périphériques sans avoir à se soucier de leur fonctionnement. Windows 2000, Windows 98, Linux, Unix, MacOS ou encore BeOS sont autant d'exemple de systèmes d'exploitation.

Presse-papiers *(p. 12)* : Egalement appelé *Clipboard* en anglais, il s'agit du mécanisme largement répendu en informatique permettant d'effectuer les très célèbres opérations de *copier-coller*.

Roster *(p. 5)* : Le Roster d'Enetshare est le carnet d'adresses de l'utilisateur. Il organise tous ses éléments sous forme d'arborescence, grâce à des familles qui peuvent contenir des contacts et aussi d'autres sous-famillles.